Int	::=	the domain of (unbounded) integer numbers, with usual operations on them
Bool	::=	the domain of Booleans
Id	::=	standard identifiers
AExp	::=	*Int*
	|	*Id*
	|	*AExp* `+` *AExp*
	|	*AExp* `/` *AExp*
BExp	::=	*Bool*
	|	*AExp* `<=` *AExp*
	|	`not` *BExp*
	|	*BExp* `and` *BExp*
Stmt	::=	`skip`
	|	*Id* `:=` *AExp*
	|	*Stmt* `;` *Stmt*
	|	`if` *BExp* `then` *Stmt* `else` *Stmt*
	|	`while` *BExp* `do` *Stmt*
Pgm	::=	`vars` **List**`{`*Id*`}` `;` *Stmt*

Figure 3.1: Syntax of IMP, a small imperative language, using algebraic BNF.

3.1 IMP: A Simple Imperative Language

To illustrate the various operational semantics styles, we have chosen a small imperative language, called IMP. The IMP language has arithmetic expressions which include the domain of arbitrarily large integer numbers, Boolean expressions, assignment statements, conditional statements, while loop statements, and sequential composition of statements. All variables used in an IMP program are expected to be declared at the beginning of the program, can only hold integer values (for simplicity, there are no Boolean variables in IMP), and are instantiated with default value 0.

3.1.1 IMP Syntax

We here define the syntax of IMP, first using the Backus-Naur form (BNF) notation for context-free grammars and then using the alternative and completely equivalent mixfix algebraic notation (see Section 2.5). The latter is in general more appropriate for semantic developments of a language.

IMP Syntax as a Context-Free Grammar

The syntax of IMP using the algebraic BNF notation is depicted in Figure 3.1. The only "algebraic" feature is the use of **List**`{`*Id*`}` for variable declarations (last production), which in this case is clear: one can declare a comma-separated list of variables. To stay more conventional in notation, we refrained from replacing the productions "*Stmt* ::= `skip` | *Stmt* `;` *Stmt*" by the algebraic production "*Stmt* ::= $\mathbf{List}^{\texttt{skip}}_{\texttt{;}}$`{`*Stmt*`}`" which captures the idea of statement sequentialization more naturally; indeed, the former yields ambiguous parsing (however, the semantics of statement sequential composition will be such that the parsing ambiguity is irrelevant, but that is not always the case).

The IMP language constructs have their usual imperative meaning. For diversity and demonstration purposes, when giving the various semantics of IMP we will assume that `+` is *non-deterministic* (it evaluates the two subexpressions in any order, possibly interleaving their corresponding evaluation steps), `/` is non-deterministic and *partial* (it will stuck the program when a division by zero takes place), `<=` is *left-right sequential* (it first evaluates the left subexpression and then the right subexpression), and that `and` is left-right sequential and *short-circuited* (it first evaluates the left subexpression and then it conditionally evaluates the right only if the left evaluated to true).

One of the main reasons for which "functional" language constructs like `+` above are allowed to be non-deterministic in language semantic definitions is because one wants to allow flexibility in how the language is implemented, not because these operations are indeed intended to have fully non-deterministic, or random, behaviors. In other words, their non-determinism is to a large extent an artifact of their intended underspecification. Some language manuals actually state explicitly that one should not rely on the order in which the arguments of language constructs are evaluated. It is typically considered to be programmers' responsibility to write their programs in such a way that one does not get different behaviors when the arguments are evaluated in different orders.

To expose some of the limitations of the existing operational approaches, in Section 3.8 we will extend IMP with expression side effects (an increment operation on variables), with abrupt termination (a halt statement), and with dynamic threads. The extension of IMP with side effects, in particular, makes the various evaluation strategies of `+`, `<=` and `and` semantically relevant.

We will assume available the domains of integers and Booleans, as well as basic operations on them which are clearly tagged (e.g., $+_{Int}$ for addition of integer numbers) to distinguish them from homonymous operations which are IMP language constructs.

We may tacitly use the following naming conventions for terms or variables throughout the remainder of this chapter: $x, X \in Id$; $a, A \in AExp$; $b, B \in BExp$; $s, S \in Stmt$; $i, I \in Int$; $t, T \in Bool$; $p, P \in Pgm$. Any of these can be primed or indexed.

IMP Syntax as an Algebraic Signature

Following the relationship between the CFG and the mixfix algebraic notations explained in Section 2.5, the BNF syntax in Figure 3.1 can be associated the entirely equivalent algebraic signature in Figure 3.2 with one (mixfix) operation per production: the terminals mixed with underscores form the name of the operation and the non-terminals give its arity. This signature is easy to define in any rewrite engine or theorem prover; moreover, it can also be defined as a data-type or corresponding structure in any programming language. We next show how it can be defined in Maude.

☆ Definition of IMP Syntax in Maude

Using the Maude notation for algebraic signatures, the algebraic signature in Figure 3.2 can yield the Maude syntax module in Figure 3.3. We have additionally picked some appropriate precedences and formatting attributes for the various language syntactic constructs.

The module `IMP-SYNTAX` in Figure 3.3 imports three "builtin" modules, namely: `INT`, which we assume it provides a sort `Int`; `BOOL`, which we assume provides a sort `Bool`; and `VAR` which we assume provides a sort `Id`. We do not give the precise definitions of these modules here, particularly because one may have many different ways to do it. In our examples from here on in the rest of the chapter we assume that `INT` contains all the integer numbers as constants of sort `Int`, that `BOOL`

sorts:

$Int,\ Bool,\ Id,\ AExp,\ BExp,\ Stmt,\ Pgm$

subsorts:

$$Int,\ Id < AExp$$
$$Bool < BExp$$

operations:

$$\_\texttt{+}\_ : AExp \times AExp \to AExp$$
$$\_\texttt{/}\_ : AExp \times AExp \to AExp$$
$$\_\texttt{<=}\_ : AExp \times AExp \to BExp$$
$$\_\texttt{and}\_ : BExp \times BExp \to BExp$$
$$\texttt{not}\_ : BExp \to BExp$$
$$\texttt{skip} : \to Stmt$$
$$\_\texttt{:=}\_ : Id \times AExp \to Stmt$$
$$\_\texttt{;}\_ : Stmt \times Stmt \to Stmt$$
$$\texttt{if}\_\texttt{then}\_\texttt{else}\_ : BExp \times Stmt \times Stmt \to Stmt$$
$$\texttt{while}\_\texttt{do}\_ : BExp \times Stmt \to Stmt$$
$$\texttt{vars}\_\texttt{;}\_ : \mathbf{List}\{Id\} \times Stmt \to Pgm$$

Figure 3.2: Syntax of IMP as an algebraic signature.

```
mod IMP-SYNTAX is including PL-INT + PL-BOOL + PL-ID .
--- AExp
  sort AExp .  subsorts Int Id < AExp .
  op _+_ : AExp AExp -> AExp [prec 33 gather (E e) format (d b o d)] .
  op _/_ : AExp AExp -> AExp [prec 31 gather (E e) format (d b o d)] .
--- BExp
  sort BExp .  subsort Bool < BExp .
  op _<=_ : AExp AExp -> BExp  [prec 37 format (d b o d)] .
  op not_ : BExp -> BExp [prec 53 format (b o d)] .
  op _and_ : BExp BExp -> BExp [prec 55 format (d b o d)] .
--- Stmt
  sort Stmt .
  op skip : -> Stmt [format (b o)] .
  op _:=_ : Id AExp -> Stmt [prec 40 format (d b o d)] .
  op _;_ : Stmt Stmt -> Stmt [prec 60 gather (e E) format (d b noi d)] .
  op if_then_else_ : BExp Stmt Stmt -> Stmt [prec 59 format (b o bni n++i bn--i n++i --)] .
  op while_do_ : BExp Stmt -> Stmt [prec 59 format (b o d n++i --)] .
--- Pgm
  sort Pgm .
  op vars_;_ : List{Id} Stmt -> Pgm [prec 70 format (nb o d ni d)] .
endm
```

Figure 3.3: IMP syntax as an algebraic signature in Maude. This definition assumes appropriate modules `INT`, `BOOL` and `VAR` defining corresponding sorts `Int`, `Bool`, and `Id`, respectively.

contains the constants `true` and `false` of sort `Bool`, and that `VAR` contains all the letters in the alphabet as constants of sort `Id`. Also, we assume that the module `INT` comes equipped with as many "builtin" operations on integers as needed.

To avoid operator name conflicts caused by Maude's operator overloading capabilities, we urge the reader *not* to use the Maude builtin `INT` and `BOOL` modules, but instead to overwrite them. Appendix A.1 shows one possible way to do this in Maude 2.5: we define new modules `INT` and `BOOL` "hooked" to the builtin integer and Boolean values but defining only a subset of operations on them and with names clearly tagged as discussed above, e.g., `_+Int_ : Int Int -> Int`, etc.

Recall from Sections 2.4 and 2.8 that lists, sets, bags, and maps are trivial algebraic structures which can be easily defined in Maude; consequently, we take the freedom to use them without definition whenever needed, as we did with using the sort `List{Id}` in Figure 3.3.

To test the syntax, one can now parse various IMP programs, such as:

```
Maude> parse
     vars n, s ;
     n := 100 ;
     s := 0 ;
     while not(n <= 0) do (
       s := s + n ;
       n := n + -1
     )
.
```

Now it is a good time to define a module, say `IMP-PROGRAMS`, containing as many IMP programs as one bears to write. Figure 3.4 shows such a module containing several IMP programs. Note that we took advantage of Maude's rewriting capabilities to save space and reuse some of the defined fragments of programs as "macros". The program `sumPgm` is calculates the sum of numbers from 1 to $n = 100$; since we do not have subtraction in IMP, we decremented the value of n by adding -1.

The program `collatzPgm` in Figure 3.4 tests Collatz' conjecture for all numbers n from 1 to $m = 10$, counting the total number of steps in s. The Collatz conjecture, still unsolved, is named after Lothar Collatz (but also known as the $3n + 1$ conjecture), who first proposed it in 1937. Take any natural number n. If n is even, divide it by 2 to get $n/2$, if n is odd multiply it by 3 and add 1 to obtain $3n + 1$. Repeat the process indefinitely. The conjecture is that no matter what number you start with, you will always eventually reach 1. Paul Erdös said about the Collatz conjecture: "Mathematics is not yet ready for such problems." While we do not attempt to solve it, we can test it even in a simple language like IMP. It is a good example program to test IMP semantics because it makes use of almost all IMP's language constructs and also has nested loops. The macro `conllatzStmt` detaches the check of a single n from the top-level loop iterating n through all $2 < n \leq m$. Note that, since we do not have multiplication and test for even numbers in IMP, we mimic them using the existing IMP constructs.

Finally, the program `countPrimesPgm` counts all the prime numbers up to m. It makes use of `primalityStmt`, which checks whether n is prime or not (writing t to 1 or to 0, respectively), and `primalityStmt` makes use of `multiplicationStmt`, which implements of fast base 2 multiplication algorithm. Defining such a module with programs helps us to test the desired language syntax (Maude will report errors if the programs that appear in the right-hand sides of the equations are not parsable), and will also help us later on to test the various semantics that we will define.

```
mod IMP-PROGRAMS is including IMP-SYNTAX .
ops sumPgm collatzPgm countPrimesPgm : -> Pgm .
ops collatzStmt multiplicationStmt primalityStmt : -> Stmt .
eq sumPgm = (
   vars n, s ;
   n := 100 ;
   while not(n <= 0) do (
     s := s + n ;
     n := n + -1
   )     ) .

eq collatzStmt = (
   while not (n <= 1) do (
     s := s + 1 ;  q := n / 2 ;  r := q + q + 1 ;
     if r <= n then n := n + n + n + 1 else n := q
   )     ) .

eq collatzPgm = (
   vars m, n, q, r, s ;
   m := 10 ;
   while not (m <= 2) do (
     n := m ;
     m := m + -1 ;
     collatzStmt
   )     ) .

eq multiplicationStmt = (        --- fast multiplication (base 2) algorithm
   z := 0 ;
   while not(x <= 0) do (
     q := x / 2 ;
     r := q + q + 1 ;
     if r <= x then z := z + y else skip ;
     x := q ;
     y := y + y
   )     ) .

eq primalityStmt = (
   i := 2 ;  q := n / i ;  t := 1 ;
   while (i <= q and 1 <= t) do (
     x := i ;
     y := q ;
     multiplicationStmt ;
     if n <= z then t := 0 else (i := i + 1 ;  q := n / i)
   )     ) .

eq countPrimesPgm = (
   vars i, m, n, q, r, s, t, x, y, z ;
   m := 10 ; n := 2 ;
   while n <= m do (
     primalityStmt ;
     if 1  <= t then s := s + 1 else skip ;
     n := n + 1
   )     ) .
endm
```

Figure 3.4: IMP programs defined in a Maude module `IMP-PROGRAMS`.

```
mod STATE is including PL-INT + PL-ID .
  sort State .
  op _|->_ : List{Id} Int -> State [prec 0] .
  op .State : -> State .
  op _&_ : State State -> State [assoc comm id: .State format(d s s d)] .
  op _(_) : State Id -> [Int] [prec 0] .        --- lookup
  op _[_/_] : State Int Id -> State [prec 0] .  --- update

  var Sigma : State .  var X X' : Id .  var Xl : List{Id} .  var I I' : Int .

  eq (Sigma & X |-> I)(X) = I .
  eq (Sigma &  X |-> I)[I' / X ] = (Sigma & X |-> I') .
  eq (X,X',Xl) |-> I = X |-> I & X' |-> I & Xl |-> I .
  eq .List{Id} |-> I = .State .
endm
```

Figure 3.5: The IMP state defined in Maude.

3.1.2 IMP State

Any operational semantics of IMP needs some appropriate notion of *state*, which is expected to map program variables to integer values. Moreover, since IMP disallows uses of undeclared variables, it suffices for the state of a given program to only map the declared variables to values and stay undefined and disallow updates of variables which were not declared.

Fortunately, all these desired IMP state operations correspond to conventional mathematical operations on *partial finite-domain functions* from variables to integers in $[Id \rightharpoonup Int]^{finite}$ (see Section 2.1.2) or, equivalently, to equationally defined structures of sort $\mathbf{Map}\{Id \mapsto Int\}$ (see Section 2.3.2 for details on the notation and the equivalence); we let *State* be an alias for the map sort above. From a rewriting logic semantic point of view, the equations defining such map structure are invisible: semantic transitions that are part of various IMP semantics will be performed *modulo* these equations. In other words, state lookup and update operations will not count as computational steps, so they will not interfere with or undesirably modify the intended computational granularity of the defined language (IMP in this case).

We let σ, σ', σ_1, etc., range over states. By defining IMP states as partial finite-domain functions $\sigma : Id \rightharpoonup Int$, we have a very natural notion of undefinedness for a variable that has not been declared and thus initialized in a state: state σ is considered *undefined* in a variable x if and only if $x \notin Dom(\sigma)$. We may use the terminology *state lookup* for the operation $\_(\_) : State \times Id \to Int$, the terminology *state update* for the operation $\_[\_/\_] : State \times Int \times Id \to State$, and the terminology *state initialization* for the operation $\_ \mapsto \_ : \mathbf{List}\{Id\} \times Int \to State$; recall from Sections 2.1.2 and 2.3.2 that the latter operation builds a partial function mapping each element in the first list argument (these elements are expected to be distinct) to the element given as second argument.

☆ Definition of IMP State in Maude

Figure 3.5 adapts the generic Maude definition of partial finite-domain functions in Figure 2.2 for our purpose here: the generic sorts `A` (for the source) and `B` (for the target) are replaced by `Id` and `Int`, respectively, and the definition of state update is simplified to only update variables that are already in the domain of the map (this suffices for IMP). The only reason for which we

bother to give this obvious module is because we want the various subsequent semantics of the IMP language, all of them including the module `STATE` in Figure 3.5, to be self-contained and executable in Maude by simply executing all the Maude code in the figures in this chapter.

3.1.3 Notes

The style that we follow in this chapter, namely to pick a simple language and then demonstrate the various language definitional approaches by means of that simple language, is quite common. In fact, we named our language IMP after a similar language introduced by Winskel in his book [81], also called IMP, which is essentially identical to ours except that it does not have variable declarations. Since most imperative languages do have variable declarations, we feel it is instructive to include them in our simple language. Winskell gives his IMP a big-step SOS, a small-step SOS, a denotational semantics, and an axiomatic semantics. Later, Nipkow [59] formalized all these semantics of IMP in the Isabelle/HOL proof assistant [60], and used it to formally relate the various semantics, effectively mechanizing most of Winskel's paper proofs; in doing so, Nipkow [59] also find several minor errors in Winskel's proofs.

Vardejo and Martì-Oliet [78, 79] show how to use Maude to implement executable semantics for several languages following both big-step and small-step SOS approaches. Like us, they also demonstrate how to define different semantics for the same simple language using different styles; they do so both for an imperative language (very similar to our IMP) and for a functional language. Șerbănuță *et al.* [72] use a similar simple imperative language to also demonstrate how to use rewriting logic to define executable semantics. In fact, this chapter is an extension of [72], both in breadth and in depth. For example, we state and prove general faithful rewriting logic representation results for each of the semantic approaches, while [72] did the same only for the particular simple imperative language considered there. Also, we cover new approaches here, such as denotational semantics, which were not covered in [78, 79, 72].